Complexity of the FIFO Stack-Up Problem*

Frank Gurski† Jochen Rethmann‡ Egon Wanke§

September 12, 2018

Abstract

We study the combinatorial FIFO stack-up problem. In delivery industry, bins have to be stacked-up from conveyor belts onto pallets with respect to customer orders. Given k sequences $q_1, \ldots, q_k$ of labeled bins and a positive integer p, the aim is to stack-up the bins by iteratively removing the first bin of one of the k sequences and put it onto an initially empty pallet of unbounded capacity located at one of p stack-up places. Bins with different pallet labels have to be placed on different pallets, bins with the same pallet label have to be placed on the same pallet. After all bins for a pallet have been removed from the given sequences, the corresponding stack-up place will be cleared and becomes available for a further pallet. The FIFO stack-up problem is to find a stack-up sequence such that all pallets can be build-up with the available p stack-up places.

In this paper, we introduce two digraph models for the FIFO stack-up problem, namely the processing graph and the sequence graph. We show that there is a processing of some list of sequences with at most p stack-up places if and only if the sequence graph of this list has directed pathwidth at most $p-1$. This connection implies that the FIFO stack-up problem is NP-complete in general, even if there are at most 6 bins for every pallet and that the problem can be solved in polynomial time, if the number p of stack-up places is assumed to be fixed. Further the processing graph allows us to show that the problem can be solved in polynomial time, if the number k of sequences is assumed to be fixed.

Keywords: computational complexity; combinatorial optimization; directed pathwidh; discrete algorithms

1 Introduction

We consider the combinatorial problem of stacking up bins from conveyor belts onto pallets. This problem originally appears in *stack-up systems* that play an important role in delivery industry and warehouses. Stack-up systems are often the back end of *order-picking systems*. A detailed description of the applied background of such systems is given in [3, 12]. Logistic experiences over 30 years lead to high flexible conveyor-based stack-up systems in delivery industry [16]. We do not intend to modify the architecture of existing systems, but try to develop efficient algorithms to control them.

The bins that have to be stacked-up onto pallets reach the stack-up system on a main conveyor belt. At the end of the line they enter the palletizing system. Here the bins are picked-up by stacker cranes or robotic arms and moved onto pallets, which are located at *stack-up places*. Often vacuum grippers are used to pick-up the bins. This picking process can be performed in different

*A short version of this paper appeared in the Proceedings of the *International Conference on Operations Research* (OR 2013) [5].

†University of Düsseldorf, Institute of Computer Science, Algorithmics for Hard Problems Group, 40225 Düsseldorf, Germany, frank.gurski@hhu.de

‡Niederrhein University of Applied Sciences, Faculty of Electrical Engineering and Computer Science, 47805 Krefeld, Germany, jochen.rethmann@hs-niederrhein.de

§University of Düsseldorf, Institute of Computer Science, Algorithms and Data Structures Group, 40225 Düsseldorf, Germany, e.wanke@hhu.de

ways depending on the architecture of the palletizing system (single-line or multi-line palletizers). Full pallets are carried away by automated guided vehicles, or by another conveyor system, while new empty pallets are placed at free stack-up places.

The developers and producers of robotic palletizers distinguish between single-line and multi-line palletizing systems. Each of these systems has its advantages and disadvantages.

In *single-line palletizing systems* there is only one conveyor belt from which the bins are picked-up. Several robotic arms or stacker cranes are placed around the end of the conveyor. We model such systems by a random access storage which is automatically replenished with bins from the main conveyor, see Figure 1. The area from which the bins can be picked-up is called the storage area. It is determined by the coverage of stacker cranes or robotic arms.

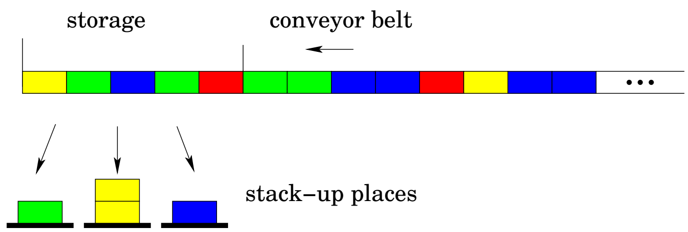

Figure 1: The single-line stack-up system using a random access storage of size 5. The colors represent the pallet labels. Bins with different colors have to be placed on different pallets, bins with the same color have to be placed on the same pallet.

In *multi-line palletizing systems* there are several buffer conveyors from which the bins are picked-up. The robotic arms or stacker cranes are placed at the end of these conveyors. Here, the bins from the main conveyor of the order-picking system first have to be distributed to the multiple infeed lines to enable parallel processing. Such a distribution can be done by some cyclic storage conveyor, see Figure 2. From the cyclic storage conveyor the bins are pushed out to the buffer conveyors. A stack-up system using a cyclic storage conveyor is, for example, located at Bertelsmann Distribution GmbH in Gütersloh, Germany. On certain days, several thousands of bins are stacked-up using a cyclic storage conveyor with a capacity of approximately 60 bins and 24 stack-up places, while up to 32 bins are destined for a pallet. This palletizing system has originally initiated our research.

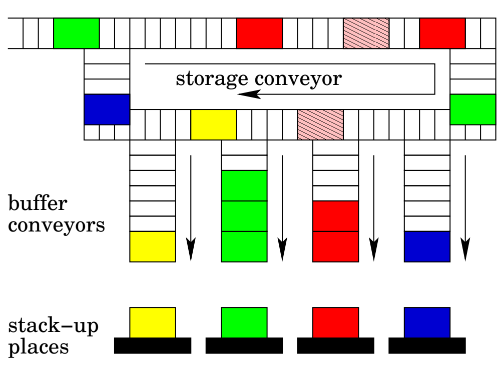

Figure 2: A multi-line stack-up system with a pre-placed cyclic storage conveyor.

If we ignore the task to distribute the bins from the main conveyor to the k buffer conveyors, i.e., if the filled buffer conveyors are already given, and if each arm can only pick-up the first bin of one of the buffer conveyors, then the system is called a FIFO palletizing system. Such systems can be modeled by several simple queues, see Figure 3.

From a theoretical point of view, an instance of the FIFO stack-up problem consists of k sequences $q_1, \ldots, q_k$ of bins and a number of available stack-up places p. Each bin of each sequence

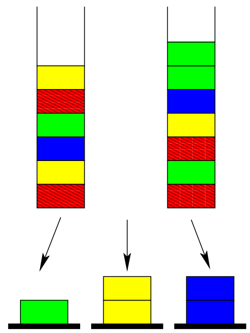

Figure 3: The FIFO stack-up system analyzed in this paper. The system is blocked (cf. page 3 for the definition), because the pallet for the red bins cannot be opened and the pallets for the green, yellow, and blue bins cannot be closed.

q_i is destined for exactly one pallet. The FIFO stack-up problem is to decide whether one can remove iteratively the bins from the sequences such that in each step only one of the first bins of $q_1, \dots, q_k$ is removed and after each step at most p pallets are open. A pallet t is called *open*, if at least one bin for pallet t has already been removed from the sequences, and if at least one bin for pallet t is still contained in the remaining sequences. If a bin b is removed from a sequence then all bins located behind b are moved-up one position to the front (cf. Section 2 for the formal definition).

Every processing should absolutely avoid blocking situations. A system is *blocked*, if all stack-up places are occupied by pallets, and non of the bins that may be used in the next step are destined for an open pallet. To unblock the system, bins have to be picked-up manually and moved to pallets by human workers. Such a blocking situation is sketched in Figure 3.

The single-line stack-up problem can be defined in the same way. An instance of the single-line stack-up problem consists of one sequence q of bins, a storage capacity s, and a number of available stack-up places p. In each step one of the first s bins of q can be removed. Everything else is defined analogously.

Many facts are known about single-line stack-up systems [12, 13, 14]. In [12] it is shown that the single-line stack-up decision problem is NP-complete, but can be solved efficiently if the storage capacity s or the number of available stack-up places p is fixed. The problem remains NP-complete as shown in [13], even if the sequence contains at most 9 bins per pallet. In [13], a polynomial-time *off-line* approximation algorithm for minimizing the storage capacity s is introduced. This algorithm yields a solution that is optimal up to a factor bounded by $\log(p)$. In [14] the performances of simple *on-line* stack-up algorithms are compared with optimal off-line solutions by a competitive analysis [2, 4].

The FIFO stack-up problem has algorithmically not been studied by other authors up to now, although stack-up systems play an important role in delivery industry and warehouses [16]. In our studies, we neither limit the number of bins for a pallet nor restrict the number of stack-up places to the number of buffer conveyors. That is, the number of stack-up places can be larger than or less than the number of buffer conveyors.

In Section 3, we introduce two digraph models that help us to find algorithmic solutions for the FIFO stack-up problem. The first digraph is the *processing graph*. It has a vertex for every possible configuration of the system and an arc from configuration A to configuration B if configuration B can be obtained from configuration A by a single processing step. If the number k of sequences is assumed to be fixed, a specific search strategy on the processing graph allows us to find an optimal solution for the FIFO stack-up problem in polynomial time. For this case, we additionally give a non-deterministic algorithm that uses only logarithmic work-space.

The second digraph is called the *sequence graph*. It has a vertex for every pallet and an arc from pallet a to pallet b, if and only if in any processing pallet b can only be closed if pallet a has already been opened. We show in Section 4 how the sequence graph can be constructed and that there is a processing of at most p stack-up places if and only if the sequence graph has directed pathwidth at most $p-1$ (cf. Section 3.2 for the formal definition of directed pathwidth). This implies that the FIFO stack-up problem can be solved in polynomial time, if the number p of given stack-up places is assumed to be fixed. In Section 5, this relationship is used to show that the FIFO stack-up problem is NP-complete in general, even if all sequences together contain at most 6 bins destined for the same pallet.

2 Preliminaries

Unless otherwise stated, k and p are some positive integers throughout the paper. We consider *sequences* $q_1 = (b_1, \dots, b_{n_1}), \dots, q_k = (b_{n_{k-1}+1}, \dots, b_{n_k})$ of pairwise distinct *bins*. These sequences represent the buffer queues (handled by the buffer conveyors) in real stack-up systems. Each bin b is labeled with a *pallet symbol* $plt(b)$ which is, without loss of generality, some positive integer. We say bin b is destined for pallet $plt(b)$. The labels of the pallets can be chosen arbitrarily, because we only need to know whether two bins are destined for the same pallet or for different pallets. In order to avoid confusion between indices (or positions) and pallet symbols, we use in our examples characters for pallet symbols. The set of all pallets of the bins in some sequence q_i is denoted by

$$plts(q_i) = \{plt(b) \mid b \in q_i\}.$$

For a list of sequences $Q = (q_1, \dots, q_k)$ we denote

$$plts(Q) = plts(q_1) \cup \dots \cup plts(q_k).$$

For some sequence $q = (b_1, \dots, b_n)$, we say bin b_i is *on the left of* bin b_j in sequence q if $i < j$. A sequence $q' = (b_j, b_{j+1}, \dots, b_n)$, $j \geq 1$, is called a *subsequence* of sequence $q = (b_1, \dots, b_n)$. We define $q - q' = (b_1, \dots, b_{j-1})$.

Let $Q = (q_1, \dots, q_k)$ and $Q' = (q'_1, \dots, q'_k)$ be two lists of sequences of bins such that each sequence q'_j, $1 \leq j \leq k$, is a subsequence of sequence q_j. Each such pair (Q, Q') is called a *configuration*. In every configuration (Q, Q') the first entry Q is the initial list of sequences of bins and the second entry Q' is the list of sequences that remain to be processed. A pallet t is called *open* in configuration (Q, Q'), if a bin of pallet t is contained in some $q'_i \in Q'$ and if another bin of pallet t is contained in some $q_j - q'_j$ for $q_j \in Q$, $q'_j \in Q'$. The *set of open pallets* in configuration (Q, Q') is denoted by $open(Q, Q')$. A pallet $t \in plts(Q)$ is called *closed* in configuration (Q, Q'), if $t \notin plts(Q')$, i.e. no sequence of Q' contains a bin for pallet t. Initially all pallets are *unprocessed*. After the first bin of a pallet t has been removed from one of the sequences, pallet t is either open or closed.

In view of the practical background, we only consider lists of sequences that together contain at least two bins for each pallet. If there is only one bin for some pallet, we can handle this bin without waiting for any further bin. Thus, we can process instances with such special pallets with at most one more stack-up place. Our definition of open pallets is useless for pallets with only one bin. Throughout the paper Q always denotes a list of some sequences of bins.

The FIFO Stack-up Problem

Consider a configuration (Q, Q'). The removal of the first bin from one subsequence $q' \in Q'$ is called *transformation step*. A sequence of transformation steps that transforms the list Q of k sequences into k empty subsequences is called a *processing* of Q.

Name: FIFO STACK-UP
Instance: A list $Q = (q_1, \dots, q_k)$ of sequences of bins, for every bin b of Q its pallet symbol $plt(b)$, and a positive integer p.
Question: Is there a processing of Q, such that in each configuration (Q, Q') during the processing at most p pallets are open?

We use the following variables: k denotes the number of sequences, and p stands for the number of stack-up places. Furthermore, m represents the number of pallets in $plts(Q)$, and n denotes the total number of bins in all sequences, i.e. $n = n_k$. Finally, $N = \max\{|q_1|, \ldots, |q_k|\}$ is the maximum sequence length. In view of the practical background, it holds $p < m$, $k < m$, $m < n$, and $N < n$.

It is often convenient to use pallet identifications instead of bin identifications to represent a sequence q. For r not necessarily distinct pallets $t_1, \ldots, t_r$ let $[t_1, \ldots, t_r]$ denote some sequence of r pairwise distinct bins $(b_1, \ldots, b_r)$, such that $plt(b_i) = t_i$ for $i = 1, \ldots, r$. We use this notion for lists of sequences as well. For the sequences $q_1 = [t_1, \ldots, t_{n_1}], \ldots, q_k = [t_{n_{k-1}+1}, \ldots, t_{n_k}]$ of pallets we define $q_1 = (b_1, \ldots, b_{n_1}), \ldots, q_k = (b_{n_{k-1}+1}, \ldots, b_{n_k})$ to be sequences of bins such that $plt(b_i) = t_i$ for $i = 1, \ldots, n_k$, and all bins are pairwise distinct.

For some list of subsequences Q' we define $front(Q')$ to be the pallets of the first bins of the queues of Q' (cf. Example 2.1 and Table 1).

Example 2.1 *Consider list $Q = (q_1, q_2)$ of sequences*

$$q_1 = (b_1, \ldots, b_4) = [a, a, b, b]$$

and

$$q_2 = (b_5, \ldots, b_{12}) = [c, d, e, c, a, d, b, e].$$

Table 1 shows a processing of Q with 3 stack-up places. The underlined bin is always the bin that will be removed in the next transformation step. We denote $Q_i = (q_1^i, q_2^i)$, thus each row represents a configuration (Q, Q_i).

i	q_1^i	q_2^i	$front(Q_i)$	remove	$open(Q, Q_i)$
0	$[a, a, b, b]$	$[\underline{c}, d, e, c, a, d, b, e]$	$\{a, c\}$	b_5	$\emptyset$
1	$[a, a, b, b]$	$[\underline{d}, e, c, a, d, b, e]$	$\{a, d\}$	b_6	$\{c\}$
2	$[a, a, b, b]$	$[\underline{e}, c, a, d, b, e]$	$\{a, e\}$	b_7	$\{c, d\}$
3	$[a, a, b, b]$	$[\underline{c}, a, d, b, e]$	$\{a, c\}$	b_8	$\{c, d, e\}$
4	$[\underline{a}, a, b, b]$	$[a, d, b, e]$	$\{a\}$	b_1	$\{d, e\}$
5	$[\underline{a}, b, b]$	$[a, d, b, e]$	$\{a\}$	b_2	$\{a, d, e\}$
6	$[b, b]$	$[\underline{a}, d, b, e]$	$\{a, b\}$	b_9	$\{a, d, e\}$
7	$[b, b]$	$[\underline{d}, b, e]$	$\{b, d\}$	b_{10}	$\{d, e\}$
8	$[b, b]$	$[\underline{b}, e]$	$\{b\}$	b_{11}	$\{e\}$
9	$[\underline{b}, b]$	$[e]$	$\{b, e\}$	b_3	$\{b, e\}$
10	$[\underline{b}]$	$[e]$	$\{b, e\}$	b_4	$\{b, e\}$
11	$[]$	$[\underline{e}]$	$\{e\}$	b_{12}	$\{e\}$
12	$[]$	$[]$	$\emptyset$	$-$	$\emptyset$

Table 1: A processing of $Q = (q_1, q_2)$ from Example 2.1 with 3 stack-up places. There is no processing of Q that needs less than 3 stack-up places.

Consider a processing of a list Q of sequences. Let $B = (b_{\pi(1)}, \ldots, b_{\pi(n)})$ be the order in which the bins are removed during the processing of Q, and let $T = (t_1, \ldots, t_m)$ be the order in which the pallets are opened during the processing of Q. Then B is called a *bin solution* of Q, and T is called a *pallet solution* of Q. The transformation in Table 1 defines the bin solution

$$B = (b_5, b_6, b_7, b_8, b_1, b_2, b_9, b_{10}, b_{11}, b_3, b_4, b_{12}),$$

and the pallet solution $T = (c, d, e, a, b)$.

During a processing of a list Q of sequences there are often configurations (Q, Q') for which it is easy to find a bin b that can be removed from Q' such that a further processing with p stack-up places is still possible. This is the case, if bin b is destined for an already open pallet, see configuration (Q, Q_3), (Q, Q_5), (Q, Q_6), (Q, Q_7), (Q, Q_9), (Q, Q_{10}), or (Q, Q_{11}) in Table 1. In the following we show:

- If one of the first bins of the sequences is destined for an already open pallet then this bin can be removed without increasing the number of stack-up places necessary to further process the sequences.

- If there is more than one bin at choice for already open pallets then the order, in which those bins are removed is arbitrary.

Both rules are useful, i.e. within the following two simple algorithms for the FIFO stack-up problem.

Algorithm 1 Generate all possible bin orders $(b_{\pi(1)}, \ldots, b_{\pi(n)})$ and verify, whether this can be a processing. This leads to a simple but very inefficient algorithm with running time $\mathcal{O}(n^2 \cdot n!)$, where $\mathcal{O}(n^2)$ time is needed for each test.

Algorithm 2 Generate all possible pallet orders $(t_{\pi(1)}, \ldots, t_{\pi(m)})$ and test, whether this can be a processing. This leads to a further simple but less inefficient algorithm with running time $\mathcal{O}(n^2 \cdot m!)$.

The running time of Algorithm 2 is much better than those of Algorithm 1 and Algorithm 2 is only possible because of the second rule. If bins for already open pallets must be removed in a special order, Algorithm 2 would not be possible.

To show the rules, consider a processing of some list Q of sequences with p stack-up places. Let

$$(b_{\pi(1)}, \ldots, b_{\pi(i-1)}, b_{\pi(i)}, \ldots, b_{\pi(\ell-1)}, b_{\pi(\ell)}, b_{\pi(\ell+1)}, \ldots, b_{\pi(n)})$$

be the order in which the bins are removed from the sequences during the processing, and let (Q, Q_j), $1 \leq j \leq n$ denote the configuration such that bin $b_{\pi(j)}$ is removed in the next transformation step. Suppose bin $b_{\pi(i)}$ will be removed in some transformation step although bin $b_{\pi(\ell)}$, $\ell > i$, for some already open pallet $plt(b_{\pi(\ell)}) \in open(Q, Q_i)$ could be removed next. We define a modified processing

$$(b_{\pi(1)}, \ldots, b_{\pi(i-1)}, b_{\pi(\ell)}, b_{\pi(i)}, \ldots, b_{\pi(\ell-1)}, b_{\pi(\ell+1)}, \ldots, b_{\pi(n)})$$

by first removing bin $b_{\pi(\ell)}$, and afterwards the bins $b_{\pi(i)}, \ldots, b_{\pi(\ell-1)}$ in the given order. Obviously, in each configuration during the modified processing there are at most p pallets open. To remove first some bin of an already open pallet is a kind of priority rule.

A configuration (Q, Q') is called a *decision configuration*, if the first bin of each sequence $q' \in Q'$ is destined for a non-open pallet, see configurations (Q, Q_0), (Q, Q_1), (Q, Q_2), (Q, Q_4), and (Q, Q_8) in Table 1, i.e. $front(Q') \cap open(Q, Q') = \emptyset$. We can restrict FIFO stack-up algorithms to deal with such decision configurations, in all other configurations the algorithms automatically remove a bin for some already open pallet.

If we have a pallet solution computed by some FIFO stack-up algorithm, we need to convert the pallet solution into a sequence of transformation steps, i.e. a processing of Q. This is done by algorithm TRANSFORM in Figure 4. Given a list of sequences $Q = (q_1, \ldots, q_k)$ and a pallet solution $T = (t_1, \ldots, t_m)$ algorithm TRANSFORM gives us in time $\mathcal{O}(n \cdot k) \subseteq \mathcal{O}(n^2)$ a bin solution of Q, i.e. a processing of Q.

Obviously, there is no other processing of Q that also defines pallet solution T but takes less stack-up places.

3 Digraph Models

Next we introduce two digraph models for the FIFO stack-up problem. To solve an instance of the FIFO stack-up problem we transform it into a digraph and by solving a graph problem we obtain a solution for the FIFO stack-up problem.

Algorithm TRANSFORM

$q'_1 := q_1, \ldots, q'_k := q_k$
$j := 1$
$T' := \{t_1\}$
repeat the following steps until $q'_1 = \emptyset, \ldots, q'_k = \emptyset$:

1. if there is a sequence q'_i such that the first bin b of q'_i is destined for a pallet in T', i.e. $plt(b) \in T'$, then remove bin b from sequence q'_i, and output b

   Comment: Bins for already open pallets are removed automatically.

2. otherwise set $j := j + 1$ and $T' := T' \cup \{t_j\}$

   Comment: If the first bin of each subsequence q'_i is destined for a non-open pallet, the next pallet of the pallet solution has to be opened.

Figure 4: Algorithm for transforming a pallet solution into a bin solution.

3.1 The processing graph

In this section we give an algorithm which computes a processing of the given sequences of bins with a minimum number of stack-up places. Such an optimal processing can always be found by computing the *processing graph* and doing some calculation on it. Before we define the processing graph let us consider some general graph problem, that will be useful to find a processing of the given sequences of bins with a minimum number of stack-up places and thus to solve the FIFO stack-up problem.

Let $G = (V, E, f)$ be a directed acyclic vertex-labeled graph. Function $f : V \to \mathbb{Z}$ assigns to every vertex $v \in V$ a value $f(v)$. Let $s \in V$ and $t \in V$ be two vertices, where s stands for source, and t stands for target. For some vertex $v \in V$ and some path $P = (v_1, \ldots, v_\ell)$ with $v_1 = s$, $v_\ell = v$ and $(v_i, v_{i+1}) \in E$ we define

$$val(P) := \max_{u \in P}(f(u))$$

to be the maximum value of f on that path. Let $\mathcal{P}_s(v)$ denote the set of all paths from vertex s to vertex v. Then we define

$$val(v) := \min_{P \in \mathcal{P}_s(v)}(val(P)).$$

If v is not reachable from s, we define $val(v) = \infty$. The problem is to compute the value $val(t)$. A solution of this problem can be found by dynamic programming and solves also the FIFO stack-up problem. The reason for this is given shortly. For some vertex $v \in V$ let

$$N^-(v) := \{u \in V \mid (u, v) \in E\}$$

be the set of predecessors of v in digraph G. The acronym N^- stands for in-neighborhood. Then it holds

$$val(v) = \max\{f(v), \min_{u \in N^-(v)}(val(u))\},$$

because each path $P \in \mathcal{P}_s(v)$ must go through some vertex $u \in N^-(v)$. That means, the value of $val(v)$ can be processed recursively. But, if we would do that, subproblems often would be solved several times. So, we use dynamic programming to solve each subproblem only once, and put these solutions together to a solution of the original problem. This is possible, since the graph is directed and acyclic.

Let $topol : V \to \mathbb{N}$ be an ordering of the vertices of digraph G such that $topol(u) < topol(v)$ holds for each $(u, v) \in E$. That means, $topol$ is a topological ordering of the vertices.

The value $val(t)$ can be computed in polynomial time by Algorithm OPT given in Figure 5. A topological ordering of the vertices of digraph G can be found by a depth first search algorithm in

Algorithm Opt

```
val[s] := f(s)
for every vertex v ≠ s in order of topol do
    val[v] := ∞
    for every u ∈ N⁻(v) do              ▷ Compute min_{u∈N⁻(v)}(val(u)) =: val(v)
        if (val[u] < val[v])
            val[v] := val[u]
            pred[v] := u
    if (val[v] < f(v))                  ▷ Compute max{f(v), val(v)}
        val[v] := f(v)
```

Figure 5: Finding an optimal processing by dynamic programming, where *topol* is a topological ordering of the vertices.

time $\mathcal{O}(|V| + |E|)$. The remaining work of algorithm Opt can also be done in time $\mathcal{O}(|V| + |E|)$. Thus, it works in time $\mathcal{O}(|V| + |E|)$, i.e. linear in the input size.

Remark 3.1 *An alternative algorithm to compute the values $val(v)$ for $v \in V$ in $G = (V, E, f)$ was suggested by an anonymous reviewer. For every $v \in V$ the value $val(v)$ can be computed as follows. Start with $r = \max_{u \in V} f(u)$. While the graph contains at least one path from s to v, we remove all vertices u with $f(u) \geq r$ and decrease r by one. Finally return $val(v) = r + 1$. The running time is not better but it shows immediately that it can be done in polynomial time. Independently we introduced a breadth first search solution for the FIFO stack-up problem in [6].*

We need some additional terms, before we show how algorithm Opt can solve the FIFO stack-up problem. For a sequence $q = (b_1, \ldots, b_n)$ let

$$left(q, i) := (b_1, \ldots, b_i)$$

denote the sequence of the first i bins of sequence q, and let

$$right(q, i) := (b_{i+1}, \ldots, b_n)$$

denote the remaining bins of sequence q after removing the first i bins. Consider again Example 2.1 and Table 1. It can be seen that a configuration is well-defined by the number of bins that are removed from each sequence. Configuration (Q, Q_6) can be described equivalently by the tuple $(2, 4)$, since in configuration (Q, Q_6) two bins of sequence q_1 have been removed, and four bins of sequence q_2 have been removed.

The position of the first bin in some sequence q_i destined for some pallet t is denoted by $first(q_i, t)$, similarly the position of the last bin for pallet t in sequence q_i is denoted by $last(q_i, t)$. For technical reasons, if there is no bin for pallet t contained in sequence q_i, then we define $first(q_i, t) = |q_i| + 1$, and $last(q_i, t) = 0$.

Example 3.2 *Consider list $Q = (q_1, q_2)$ of the sequences*

$$q_1 = [a, b, c, a, b, c]$$

and

$$q_2 = [d, e, f, d, e, f, a, b, c].$$

Then we get $left(q_1, 2) = [a, b]$, $right(q_1, 2) = [c, a, b, c]$, $left(q_2, 3) = [d, e, f]$, and $right(q_2, 3) = [d, e, f, a, b, c]$.

If we denote $q'_1 := right(q_1, 2)$, and $q'_2 := right(q_2, 3)$, then there are 5 pallets open in configuration (Q, Q') with $Q' = (q'_1, q'_2)$, namely the pallets a, b, d, e, and f.

We generalize this for a list $Q = (q_1, \dots, q_k)$ of sequences and we define the cut

$$\begin{aligned} cut_Q(i_1, \dots, i_k) \ := \ & \{t \in plts(Q) \mid \\ & \exists j, j', b \in left(q_j, i_j), b' \in right(q_{j'}, i_{j'}) : plt(b) = plt(b') = t\} \end{aligned}$$

at some configuration $(i_1, \dots, i_k)$ to be the set of pallets t such that one bin for pallet t has already been removed and another bin for pallet t is still contained in some sequence. Let $\#cut_Q(i_1, \dots, i_k)$ be the number of elements in $cut_Q(i_1, \dots, i_k)$.

The intention of a processing graph $G = (V, E, f)$ is the following. Suppose each vertex $v \in V$ of digraph G represents a configuration $(i_1, \dots, i_k)$ during the processing of some set of sequences Q. Suppose further, an arc $(u, v) \in E$ represents a transformation step during this processing, such that a bin b is removed from some sequence in configuration u resulting in configuration v. Suppose also that each vertex v is assigned the number of open pallets in configuration v, i.e. number $\#cut_Q(i_1, \dots, i_k)$. If vertex s represents the initial configuration $(0, 0, \dots, 0)$, while vertex t represents the final configuration $(|q_1|, |q_2|, \dots, |q_k|)$, then we are searching a path P from s to t such that the maximal number on path P is minimal. Thus, an optimal processing of Q can be found by Algorithm Opt given in Figure 5.

The *processing graph* has a vertex for each possible configuration. Each vertex v of the processing graph is labeled by the vector $h(v) = (v_1, \dots, v_k)$, where v_i denotes the position of the bin that has been removed last from sequence q_i. Further each vertex v of the processing graph is labeled by the value $f(v) = \#cut_Q(v_1, \dots, v_k) = \#cut_Q(h(v))$. There is an arc from vertex u labeled by $(u_1, \dots, u_k)$ to vertex v labeled by $(v_1, \dots, v_k)$ if and only if $u_i = v_i - 1$ for exactly one element of the vector and for all other elements of the vector $u_j = v_j$. The arc is labeled with the pallet symbol of that bin, that will be removed in the corresponding transformation step. That means, the processing graph represents all the possible branchings through the decision process. For the sequences of Example 3.2 we get the processing graph of Figure 6. The processing graph is directed and acyclic, and we use this digraph to compute the values of $\#cut_Q(i_1, \dots, i_k)$ iteratively in the following way.

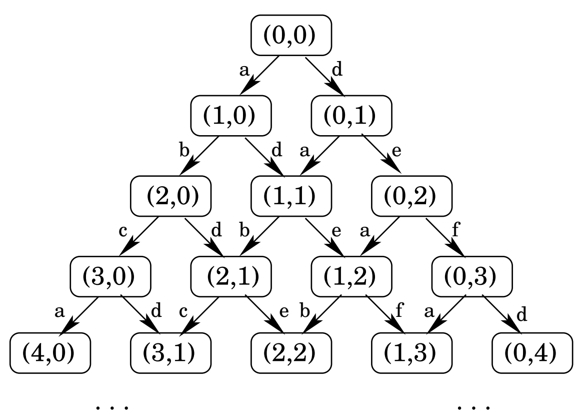

Figure 6: The processing graph of Example 3.2. For every vertex v the vector $h(v)$ in the vertex represents the configuration of v and every arc is labeled with the pallet symbol of that bin, that is removed in the corresponding transformation step.

First, since none of the bins has been removed from some sequence, we have $\#cut_Q(0, \dots, 0) = 0$. Since the processing graph is directed and acyclic, there exists a topological ordering *topol* of the vertices. The vertices are processed according to the order *topol*. In each transformation step we remove exactly one bin for some pallet t from some sequence q_j, thus

$$\begin{aligned} & \#cut_Q(i_1, \dots, i_{j-1}, i_j + 1, i_{j+1}, \dots, i_k) \\ & \quad = \ \#cut_Q(i_1, \dots, i_{j-1}, i_j, i_{j+1}, \dots, i_k) \ + \ c_j \end{aligned}$$

where $c_j = 1$ if pallet t has been opened in the transformation step, and $c_j = -1$ if pallet t has been closed in the transformation step. Otherwise, c_j is zero. If we put this into a formula we get

$$c_j = \begin{cases} 1, & \text{if } \mathit{first}(q_j, t) = i_j + 1 \text{ and } \mathit{first}(q_\ell, t) > i_\ell \ \forall\, \ell \neq j \\ -1, & \text{if } \mathit{last}(q_j, t) = i_j + 1 \text{ and } \mathit{last}(q_\ell, t) \leq i_\ell \ \forall\, \ell \neq j \\ 0, & \text{otherwise.} \end{cases}$$

Please remember our technical definition of $\mathit{first}(q,t)$ and $\mathit{last}(q,t)$ from page 8 for the case that $t \notin \mathit{plts}(q)$.

That means, the calculation of value $\#\mathit{cut}_Q(i_1, \ldots, i_k)$ for the vertex labeled $(i_1, \ldots, i_k)$ depends only on already calculated values. Figure 7 shows such a processing for the sequences of Example 3.2.

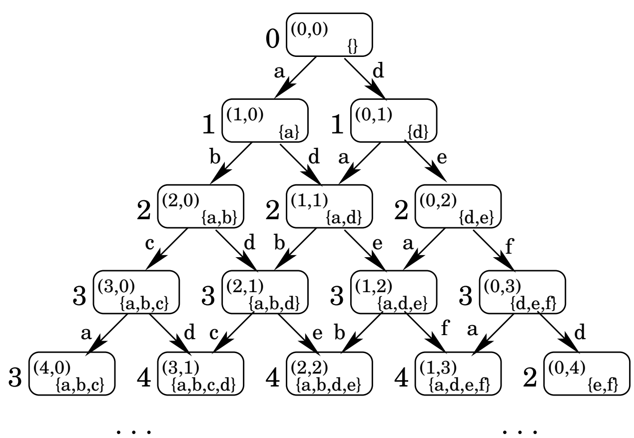

Figure 7: The processing graph of Example 3.2. For every vertex v the set in the vertex v shows the open pallets, the vector $h(v)$ in the vertex represents the configuration of v, and the number $f(v)$ on the left of the vertex represents the number of open pallets in the corresponding configuration $h(v)$.

To efficiently perform the processing of c_j, we have to store for each pallet the first and last bin in each sequence. Table 2 shows these values for the sequences of Example 3.2. We compute all the values $\mathit{first}(q_j, t)$ and $\mathit{last}(q_j, t)$ for $q_i \in Q$ and $t \in \mathit{plts}(Q)$ in some preprocessing phase. Such a preprocessing can be done in time $\mathcal{O}(k \cdot N + k \cdot m)$, which can be bounded due to $m \leq k \cdot N$ by $\mathcal{O}(k \cdot (N+1)^k)$. Then the calculation of value $\#\mathit{cut}_Q(h(v))$ for the vertex v representing configuration $h(v)$ can be done in time $\mathcal{O}(k)$.

pallet t	a	b	c	d	e	f
$\mathit{first}(q_1, t)$	1	2	3	7	7	7
$\mathit{first}(q_2, t)$	7	8	9	1	2	3
$\mathit{last}(q_1, t)$	4	5	6	0	0	0
$\mathit{last}(q_2, t)$	7	8	9	4	5	6

Table 2: The values $\mathit{first}(q_j, t)$ and $\mathit{last}(q_j, t)$ for $q_i \in Q$ and $t \in \mathit{plts}(Q)$ of Example 3.2.

The computation of all at most $(N+1)^k$ values $\#\mathit{cut}_Q(i_1, \ldots, i_k)$ can be performed in time $\mathcal{O}(k \cdot (N+1)^k)$. After that we can use Algorithm Opt to compute the minimal number of stack-up places necessary to process the given FIFO stack-up problem in time $\mathcal{O}(|V|+|E|) \subseteq \mathcal{O}(k \cdot (N+1)^k)$, since $|V| \leq (N+1)^k$ and $|E| \leq k \cdot |V|$. If the size of the processing graph is polynomially bounded in the size of the input, the FIFO stack-up problem can be solved in polynomial time. In general the processing graph can be too large to efficiently solve the problem.

3.2 The Sequence Graph

Next we show a correlation between the used number of stack-up places for a processing of an instance Q and the directed pathwidth of a digraph G_Q defined by Q. The notion of *directed pathwidth* was introduced by Reed, Seymour, and Thomas in the 1990s and leads a restriction of *directed treewidth* which is defined by Johnson, Robertson, Seymour, and Thomas in [7].

This correlation implies that 1.) the FIFO stack-up problem is NP-complete and 2.) a pallet solution can be computed in polynomial time if there are only a fixed number of stack-up places.

A *directed path-decomposition* of a digraph $G = (V, E)$ is a sequence

$$(X_1, \dots, X_r)$$

of subsets of V, called *bags*, that satisfy the following three properties.

(dpw-1) $X_1 \cup \ldots \cup X_r \;=\; V$

(dpw-2) for each arc $(u, v) \in E$ there are indices i, j with $i \leq j$ such that $u \in X_i$ and $v \in X_j$

(dpw-3) if $u \in X_i$ and $u \in X_j$ for some node u and two indices i, j with $i \leq j$, then $u \in X_\ell$ for all indices ℓ with $i \leq \ell \leq j$

The *width* of a directed path-decomposition $\mathcal{X} = (X_1, \dots, X_r)$ is

$$\max_{1 \leq i \leq r} |X_i| - 1.$$

The *directed pathwidth* of G, d-pw(G) for short, is the smallest integer w such that there is a directed path-decomposition for G of width w. For symmetric digraphs, the directed pathwidth is equivalent to the undirected pathwidth of the corresponding undirected graph [9], which implies that determining whether the pathwidth of some given digraph is at most some given value w is NP-complete. For each fixed integer w, it is decidable in polynomial time whether a given digraph has directed pathwidth at most w, see Tamaki [15].

The *sequence graph* $G_Q = (V, E)$ for an instance $Q = (q_1, \dots, q_k)$ of the FIFO stack-up problem is defined by vertex set $V = \mathit{plts}(Q)$ and the following set of arcs. There is an arc $(u, v) \in E$ if and only if there is some sequence where a bin destinated for u is on the left of some bin destinated for v.

More formally, there is an arc $(u, v) \in E$ if and only if there is a sequence $q_i = (b_{n_{i-1}+1}, \dots, b_{n_i})$ with two bins b_{j_1}, b_{j_2} such that (1) $j_1 < j_2$, (2) $\mathit{plt}(b_{j_1}) = u$, (3) $\mathit{plt}(b_{j_2}) = v$, and (4) $u \neq v$.

Example 3.3 *Figure 8 shows the sequence graph G_Q for $Q = (q_1, q_2, q_3)$ with sequences $q_1 = [a, a, d, e, d]$, $q_2 = [b, b, d]$, and $q_3 = [c, c, d, e, d]$.*

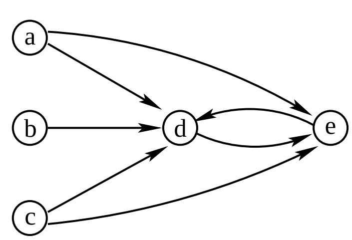

Figure 8: Sequence graph G_Q of Example 3.3.

If $G_Q = (V, E)$ has an arc $(u, v) \in E$ then $u \neq v$ and for every processing of Q, pallet u is opened before pallet v is closed. Digraph $G_Q = (V, E)$ can be computed in time $\mathcal{O}(n+k\cdot|E|) \subseteq \mathcal{O}(n+k\cdot m^2)$ by the algorithm Create Sequence Graph shown in Figure 9.

A value is added to a list only if it is not already contained. To check this efficiently in time $\mathcal{O}(1)$ we have to use a boolean array. In our algorithm V and L are realized by boolean arrays.

Algorithm CREATE SEQUENCE GRAPH

for each sequence $q \in Q$ do
  $b :=$ first bin of sequence q
  add $plt(b)$ to vertex set V, if it is not already contained
  $L := (plt(b))$ $\triangleright$ L contains pallets of bins up to bin b
  for $i := 2$ to $|q|$ do
    $b := i$-th bin of sequence q
    add $plt(b)$ to vertex set V, if it is not already contained
    if $(i == last(q, plt(b)))$
      for each pallet $t \in L$ do
        if $t \neq plt(b)$ add arc $(t, plt(b))$ to arc set A, if it is not already contained
    if $(i == first(q, plt(b)))$
      $append(plt(b), L)$

Figure 9: Create the sequence graph $G = (V, A)$ for some given list of sequences Q.

Therefore we need some preprocessing phase where we run through each sequence and seek for the pallets. This can be done in time $\mathcal{O}(n + k \cdot m) \subseteq \mathcal{O}(n + m^2)$.

The following two Theorems show the correlation between the used number of stack-up places for a processing of an instance Q and the directed pathwidth of the sequence graph G_Q.

First we want to emphasize that not every directed path-decomposition of a sequence graph G_Q immediately leads to a pallet solution. In our Example 3.3 the sequence $(\{e, a\}, \{e, b\}, \{e, c\}, \{e, d\})$ is a directed path-decomposition of optimal width 1 for the sequence graph G_Q. But opening the pallets one after another leads to (e, a, b, c, d), which is no pallet solution since pallet e cannot be opened at first and must be put on hold. Within the proof of Theorem 3.5 we show how to transform a directed path-decomposition of G_Q into a pallet solution for Q. Example 3.6 and Table 3 illustrate this process.

Theorem 3.4 *A processing $(Q, Q_0), (Q, Q_1), \ldots, (Q, Q_n)$ of Q with $Q_0 = Q$, $Q_n = (\emptyset, \ldots, \emptyset)$, and p stack-up places defines a directed path-decomposition*

$$\mathcal{X} = (open(Q, Q_0), \ldots, open(Q, Q_n))$$

for G_Q of width $p - 1$.

Proof We show that

$$\mathcal{X} = (open(Q, Q_0), \ldots, open(Q, Q_n))$$

satisfies all properties of a directed path-decomposition.

(dpw-1) $open(Q, Q_0) \cup \cdots \cup open(Q, Q_n) = plts(Q)$, because every pallet is opened at least once during a processing.

(dpw-2) If $(u, v) \in E$ then there are indices i, j with $i \leq j$ such that $u \in open(Q, Q_i)$ and $v \in open(Q, Q_j)$, because v can not be closed before u is opened.

(dpw-3) If $u \in open(Q, Q_i)$ and $u \in open(Q, Q_j)$ for some pallet u and two indices i, j with $i \leq j$, then $u \in open(Q, Q_\ell)$ for all indices ℓ with $i \leq \ell \leq j$, because every pallet is opened at most once.

Since Q is processed with p stack-up places, we have $|open(Q, Q_i)| \leq p$ for $0 \leq i \leq n$, and therefore $\mathcal{X}$ has width at most $p - 1$. □

Theorem 3.5 *If there is a path-decomposition $\mathcal{X} = (X_1, \ldots, X_r)$ for G_Q of width $p-1$ then there is a processing of Q with p stack-up places.*

Proof For a pallet t let $\alpha(\mathcal{X}, t)$ be the smallest i such that $t \in X_i$ and $\beta(\mathcal{X}, t)$ be the largest i such that $t \in X_i$, see Example 3.6. Then $t \in X_i$ if and only if $\alpha(\mathcal{X}, t) \leq i \leq \beta(\mathcal{X}, t)$. If (t_1, t_2) is an arc of G_Q, then $\alpha(\mathcal{X}, t_1) \leq \beta(\mathcal{X}, t_2)$. This follows by (dpw-2) of the definition of a directed path-decomposition.

Instance Q can be processed as follows. If it is necessary to open in a configuration (Q, Q') a new pallet, then we open a pallet t of *front*(Q') for which $\alpha(\mathcal{X}, t)$ is minimal.

We next show that for every configuration (Q, Q') of the processing above there is a bag X_i in the directed path-decomposition $\mathcal{X} = (X_1, \ldots, X_r)$, such that *open*$(Q, Q') \subseteq X_i$. This implies that the processing uses at most p stack-up places.

First, let (Q, Q') be a decision configuration, let $t \in$ *front*(Q') such that $\alpha(\mathcal{X}, t)$ is minimal, and let $t' \in$ *open*(Q, Q') be an already open pallet. We show that the intervals $[\alpha(\mathcal{X}, t), \beta(\mathcal{X}, t)]$ and $[\alpha(\mathcal{X}, t'), \beta(\mathcal{X}, t')]$ overlap, because $\alpha(\mathcal{X}, t) \leq \beta(\mathcal{X}, t')$ and $\alpha(\mathcal{X}, t') \leq \beta(\mathcal{X}, t)$.

(1) To show $\alpha(\mathcal{X}, t) \leq \beta(\mathcal{X}, t')$ we observe the following:

Since $t' \notin$ *front*(Q'), there has to be a pallet $t'' \in$ *front*(Q') such that (t'', t') is an arc of G_Q. This implies that $\alpha(\mathcal{X}, t'') \leq \beta(\mathcal{X}, t')$. Since t is a pallet of *front*(Q') for which $\alpha(\mathcal{X}, t)$ is minimal, we have $\alpha(\mathcal{X}, t) \leq \alpha(\mathcal{X}, t'')$ and thus $\alpha(\mathcal{X}, t) \leq \beta(\mathcal{X}, t')$.

(2) To show $\alpha(\mathcal{X}, t') \leq \beta(\mathcal{X}, t)$ we observe the following:

Let (Q, Q'') be the configuration in which t' has been opened.

(a) $t \in$ *front*(Q''). Since t' is a pallet of *front*(Q'') for which $\alpha(\mathcal{X}, t')$ is minimal, we have $\alpha(\mathcal{X}, t') \leq \alpha(\mathcal{X}, t)$ and thus $\alpha(\mathcal{X}, t') \leq \beta(\mathcal{X}, t)$.

(b) $t \notin$ *front*(Q''). Then there is a palled $t'' \in$ *front*(Q'') such that (t'', t) is an arc of G_Q. This implies that $\alpha(\mathcal{X}, t'') \leq \beta(\mathcal{X}, t)$. Since t' is a pallet of *front*(Q'') for which $\alpha(\mathcal{X}, t')$ is minimal, we have $\alpha(\mathcal{X}, t') \leq \alpha(\mathcal{X}, t'') \leq \beta(\mathcal{X}, t)$.

Finally, let $(Q, \hat{Q})$ be an arbitrary configuration during the processing of Q. By the discussion above, we can conclude that for every pair $t_i, t_j \in$ *open*$(Q, \hat{Q})$ the intervals $[\alpha(\mathcal{X}, t_i), \beta(\mathcal{X}, t_i)]$ and $[\alpha(\mathcal{X}, t_j), \beta(\mathcal{X}, t_j)]$ overlap, because t_i is opened before t_j or vice versa. Since the cut of all intervals $[\alpha(\mathcal{X}, t_i), \beta(\mathcal{X}, t_i)]$, $t_i \in$ *open*$(Q, \hat{Q})$ mutually overlap, there is a bag X_j in the directed path-decomposition $\mathcal{X}$ such that *open*$(Q, \hat{Q}) \subseteq X_j$. □

Example 3.6 *We consider digraph G_Q for $Q = (q_1, q_2, q_3)$ with sequences $q_1 = [a, a, d, e, d]$, $q_2 = [b, b, d]$, and $q_3 = [c, c, d, e, d]$ from Example 3.3. Sequence $(\{a, e\}, \{b, e\}, \{c, e\}, \{d, e\})$ is a directed path-decomposition of width 1, which implies the following values for α and β used in the proof of Theorem 3.5.*

pallet t	*a*	*b*	*c*	*d*	*e*
$\alpha(\mathcal{X}, t)$	1	2	3	4	1
$\beta(\mathcal{X}, t)$	1	2	3	4	4

Table 3 shows a processing of Q with 2 stack-up places and pallet solution $S = (a, b, c, d, e)$. The underlined bin is always the bin that will be removed in the next transformation step. We denote $Q_i = (q_1^i, q_2^i, q_3^i)$, thus each row represents a configuration (Q, Q_i).

4 Hardness Result

Next we will show the hardness of the FIFO stack-up problem. In contrast to Section 3 we will transform an instance of a graph problem into an instance of the FIFO stack-up problem.

Let $G = (V, E)$ be a digraph. We will assume that $G = (V, E)$ does not contain any vertex with only outgoing arcs and not contain any vertex with only incoming arcs. This is only for technical reasons and the removal of such vertices will not change the directed pathwidth of G, because a

i	q_1^i	q_2^i	q_3^i	$front(Q')$	$open(Q, Q_i)$
0	$[\underline{a}, a, d, e, d]$	$[b, b, d]$	$[c, c, d, e, d]$	$\{a, b, c\}$	$\emptyset$
1	$[\underline{a}, d, e, d]$	$[b, b, d]$	$[c, c, d, e, d]$	$\{a, b, c\}$	$\{a\}$
2	$[d, e, d]$	$[\underline{b}, b, d]$	$[c, c, d, e, d]$	$\{b, c, d\}$	$\emptyset$
3	$[d, e, d]$	$[\underline{b}, d]$	$[c, c, d, e, d]$	$\{b, c, d\}$	$\{b\}$
4	$[d, e, d]$	$[d]$	$[\underline{c}, c, d, e, d]$	$\{c, d\}$	$\emptyset$
5	$[d, e, d]$	$[d]$	$[\underline{c}, d, e, d]$	$\{c, d\}$	$\{c\}$
6	$[\underline{d}, e, d]$	$[d]$	$[d, e, d]$	$\{d\}$	$\emptyset$
7	$[e, d]$	$[\underline{d}]$	$[d, e, d]$	$\{d, e\}$	$\{d\}$
8	$[e, d]$	$[]$	$[\underline{d}, e, d]$	$\{d, e\}$	$\{d\}$
9	$[\underline{e}, d]$	$[]$	$[e, d]$	$\{e\}$	$\{d\}$
10	$[\underline{d}]$	$[]$	$[e, d]$	$\{d, e\}$	$\{d, e\}$
11	$[]$	$[]$	$[\underline{e}, d]$	$\{e\}$	$\{d, e\}$
12	$[]$	$[]$	$[\underline{d}]$	$\{d\}$	$\{d\}$
13	$[]$	$[]$	$[]$	$\emptyset$	$\emptyset$

Table 3: A processing of Q with respect to a given directed path-decomposition for G_Q of Example 3.6.

vertex u with only outgoing arcs can be placed in a singleton $X_i = \{u\}$ at the beginning of the directed path-decomposition and a vertex u with only incoming arcs can be placed in a singleton $X_i = \{u\}$ at the end of the directed path-decomposition, without to change its width.

Let $G = (V, E)$ be some digraph and $E = \{e_1, \ldots, e_\ell\}$ its arc set. The *queue system* $Q_G = (q_1, \ldots, q_\ell)$ for G is defined as follows.

(1) There are 2ℓ bins $b_1, \ldots, b_{2\ell}$.

(2) Queue $q_i = (b_{2i-1}, b_{2i})$ for $1 \leq i \leq \ell$.

(3) The pallet symbol of bin b_{2i-1} is the first vertex of arc e_i and the pallet symbol of b_{2i} is the second vertex of arc e_i for $1 \leq i \leq \ell$. Thus $plts(Q_G) = V$.

The definition of queue system Q_G and sequence graph G_Q, defined in Section 3.2, now imply the following proposition.

Proposition 4.1 *For every digraph G,*

$$G \quad = \quad G_{Q_G}.$$

Example 4.2 *Consider the digraph G of Figure 10. The corresponding queue system is $Q_G = (q_1, q_2, q_3, q_4, q_5, q_6, q_7)$, where*

$$\begin{array}{llllllll} q_1 = [a, b], & q_2 = [b, c], & q_3 = [c, d], & q_4 = [d, e], \\ q_5 = [e, a], & q_6 = [e, f], & q_7 = [f, a]. \end{array}$$

The sequence graph of Q_G is G.

Lemma 4.3 *There is a directed path-decomposition for G of width $p - 1$ if and only if there is a processing of Q_G with at most p stack-up places.*

Proof By Proposition 4.1 we know that $G = G_{Q_G}$. If there is a directed path-decomposition for $G = G_{Q_G}$ of width $p - 1$ then by Theorem 3.5 there is a processing of Q_G with at most p stack-up places. If there is a processing of Q_G with at most p stack-up places then by Theorem 3.4 there is directed path-decomposition for G of width $p - 1$. $\square$

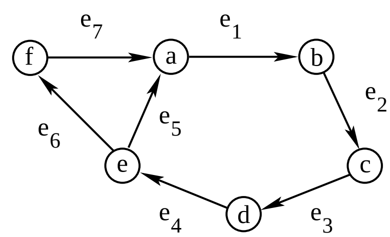

Figure 10: Digraph G of Example 4.2.

Theorem 4.4 *Given a list $Q = (q_1, \ldots, q_k)$ of k sequences of bins and some positive integer p. The problem to decide whether there is a processing of Q with at most p stack-up places is NP-complete.*

Proof The given problem is obviously in NP. Determining whether the pathwidth of some given (undirected) graph is at most some given value w is NP-complete [8] and for symmetric digraphs a special case of the problem on directed graphs (cf. Introduction of [9]). Thus the NP-hardness follows from Lemma 4.3, because Q_G can be constructed from G in linear time. □

By the results of [1] it is shown in [10] that determining whether the pathwidth of some given (undirected) graph G is at most some given value w remains NP-complete even for planar graphs with maximum vertex degree 3. Thus the problem to decide, whether the directed pathwidth of some given symmetric digraph G is at most some given value w remains NP-complete even for planar digraphs with maximum vertex in-degree 3 and maximum vertex out-degree 3. Thus, by our transformation of graph G into Q_G we get sequences that contain together at most 6 bins per pallet. Hence, the FIFO stack-up problem is NP-complete even if the number of bins per pallet is bounded. Therefore we have proved the following statement.

Corollary 4.5 *Given a list $Q = (q_1, \ldots, q_k)$ of k sequences of bins and some positive integer p. The problem to decide whether there is a processing of Q with at most p stack-up places is NP-complete, even if the sequences of Q contain together at most 6 bins per pallet.*

5 Bounded FIFO stack-up systems

In this section we show that the FIFO stack-up problem can be solved in polynomial time, if the number k of sequences or the number p of stack-up places is assumed to be fixed.

5.1 Fixed number of sequences

In this section we assume that the number k of sequences is fixed.

In Section 3.1 we have shown that the FIFO stack-up problem can be solved by dynamic programming using the processing graph in time $\mathcal{O}(k \cdot (N+1)^k)$. Thus, we have already shown the following result.

Theorem 5.1 *Given a list $Q = (q_1, \ldots, q_k)$ of sequences of bins for some fixed k and a number of stack-up places p. The question whether the sequences can be processed with at most p stack-up places can be solved in polynomial time.*

Next we improve this result.

Theorem 5.2 *Given a list $Q = (q_1, \ldots, q_k)$ of sequences of bins for some fixed k and a number of stack-up places p. The question whether the sequences can be processed with at most p stack-up places is non-deterministically decidable using logarithmic work-space.*

Proof We need $k+1$ variables, namely $pos_1, \ldots, pos_k$ and *open*. Each variable pos_i is used to store the position of the bin which has been removed last from sequence q_i. Variable *open* is used to store the number of open pallets. These variables take $(k+1) \cdot \lceil \log(n) \rceil$ bits. The simulation starts with $pos_1 := 0, \ldots, pos_k := 0$ and $open := 0$.

(i) Choose non-deterministically any index i and increment variable pos_i. Let b be the bin on position pos_i in sequence q_i, and let $t := plt(b)$ be the pallet symbol of bin b.
Comment: The next bin b from some sequence q_i will be removed.

(ii) If $first(q_j, t) > pos_j$ or $t \notin plts(q_j)$ for each $j \neq i$, $1 \leq j \leq k$, and $first(q_i, t) = pos_i$, then increment variable *open*.
Comment: If the removed bin b was the first bin of pallet t that ever has been removed from any sequence, then pallet t has just been opened.

(iii) If $last(q_j, t) \leq pos_j$ or $t \notin plts(q_j)$ for each j, $1 \leq j \leq k$, then decrement variable *open*.
Comment: If bin b was the last one of pallet t, then pallet t has just been closed.

If *open* is set to a value greater than p in Step (ii) of the algorithm then the execution is stopped in a non-accepting state. To execute Steps (ii) and (iii) we need a fixed number of additional variables. Thus, all steps can be executed non-deterministically using logarithmic work-space. □

Theorem 5.2 implies that the FIFO stack-up problem with a fixed number of given sequences can be solved in polynomial time since NL is a subset of P. The class NL is the set of problems decidable non-deterministically on logarithmic work-space. For example, reachability of nodes in a given graph is NL-complete, see [11]. Even more, it can be solved in parallel in polylogarithmic time with polynomial amount of total work, since NL is a subset of NC_2. The class NC_2 is the set of problems decidable in time $\mathcal{O}(\log^2(n))$ on a parallel computer with a polynomial number of processors, see [11].

5.2 Fixed number of stack-up places

In [15] it is shown that the problem of determining the bounded directed pathwidth of a digraph is solvable in polynomial time. By Theorem 3.4 and Theorem 3.5 the FIFO stack-up problem with fixed number p of stack-up places is also solvable in polynomial time.

Theorem 5.3 *Given a list $Q = (q_1, \ldots, q_k)$ of k sequences of bins and a fixed number of stack-up places p. The question whether the sequences can be processed with at most p stack-up places can be solved in polynomial time.*

6 Conclusions and Outlook

In this paper, we have shown that the FIFO stack-up problem is NP-complete in general, even if in all sequences together there are at most 6 bins destined for the same pallet. The problem can be solved in polynomial time, if the number k of sequences or if the number p of stack-up places is assumed to be fixed.

In our future work, we want to find an answers to the following questions. Our hardness result motivates to analyze the time complexity of the FIFO stack-up problem if in all sequences together there are at most c, $1 < c < 6$, bins destined for the same pallet.

Further we are interested in online algorithms for instances where we only know the first c bins of every sequence instead of the complete sequences. Especially, we are interested in the answer to the following question: Is there a d-competitive online algorithm? Such an algorithm must compute a processing of some Q with at most $p \cdot d$ stack-up places, if Q can be processed with at most p stack-up places.

In real life the bins arrive at the stack-up system on the main conveyor of a pick-to-belt orderpicking system. That means, the distribution of bins to the sequences has to be computed.

Up to now we consider the distribution as given. We intend to consider how to compute an optimal distribution of the bins from the main conveyor onto the sequences such that a minimum number of stack-up places is necessary to stack-up all bins from the sequences.

Acknowledgements

We thank Prof. Dr. Christian Ewering who shaped our interest into controlling stack-up systems. Furthermore, we are very grateful to Bertelsmann Distribution GmbH in Gütersloh, Germany, for providing the possibility to get an insight into the real problematic nature and for providing real data instances.

References

[1] H.L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–23, 1993.

[2] A. Borodin. *On-line Computation and Competitive Analysis*. Cambridge University Press, 1998.

[3] R. de Koster. Performance approximation of pick-to-belt orderpicking systems. *European Journal of Operational Research*, 92:558–573, 1994.

[4] A. Fiat and G.J. Woeginger. *Online Algorithms: The state of the art*, volume 1442 of *LNCS*. Springer-Verlag, 1998.

[5] F. Gurski, J. Rethmann, and E. Wanke. Moving bins from conveyor belts onto pallets using FIFO queues. In *Proceedings of the International Conference on Operations Research (OR 2013), Selected Papers*, pages 185–191. Springer-Verlag, 2014.

[6] F. Gurski, J. Rethmann, and E. Wanke. A practical approach for the FIFO stack-up problem. In *Modelling, Computation and Optimization in Information Systems and Management Sciences*, volume 360 of *Advances in Intelligent Systems and Computing*, pages 141–152. Springer, 2015.

[7] T. Johnson, N. Robertson, P.D. Seymour, and R. Thomas. Directed tree-width. *Journal of Combinatorial Theory, Series B*, 82:138–155, 2001.

[8] T. Kashiwabara and T. Fujisawa. NP-completeness of the problem of finding a minimum-clique-number interval graph containing a given graph as a subgraph. In *Proceedings of the International Symposium on Circuits and Systems*, pages 657–660, 1979.

[9] K. Kitsunai, Y. Kobayashi, K. Komuro, H. Tamaki, and T. Tano. Computing directed pathwidth in $O(1.89^n)$ time. In *Proceedings of International Workshop on Parameterized and Exact Computation*, volume 7535 of *LNCS*, pages 182–193. Springer-Verlag, 2012.

[10] B. Monien and I.H. Sudborough. Min cut is NP-complete for edge weighted trees. *Theoretical Computer Science*, 58:209–229, 1988.

[11] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, New York, 1994.

[12] J. Rethmann and E. Wanke. Storage controlled pile-up systems, theoretical foundations. *European Journal of Operational Research*, 103(3):515–530, 1997.

[13] J. Rethmann and E. Wanke. On approximation algorithms for the stack-up problem. *Mathematical Methods of Operations Research*, 51:203–233, 2000.

[14] J. Rethmann and E. Wanke. Stack-up algorithms for palletizing at delivery industry. *European Journal of Operational Research*, 128(1):74–97, 2001.

[15] H. Tamaki. A Polynomial Time Algorithm for Bounded Directed Pathwidth. In *Proceedings of Graph-Theoretical Concepts in Computer Science*, volume 6986 of *LNCS*, pages 331–342. Springer-Verlag, 2011.

[16] K.L. Yam, editor. *The Wiley Encyclopedia of Packaging Technology*. John Wiley & Sons, New York, 2009.